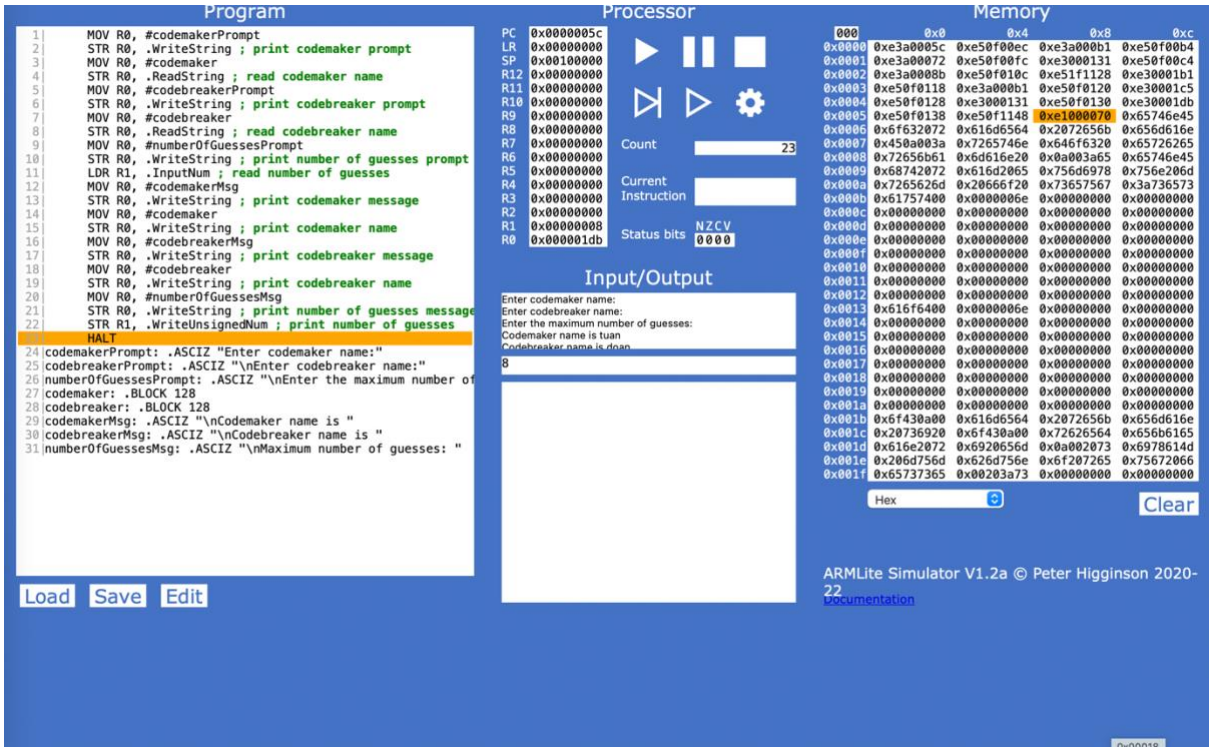
COMPUTER SYSTEM ASSIGNMENT 2 REPORT

Student Name: Anh Tuan Doan
Student ID: 103526745
Unit code: COS10004
Lab session: Friday 8:30am-10:30am at EN409

Table of Contents

Stage 1

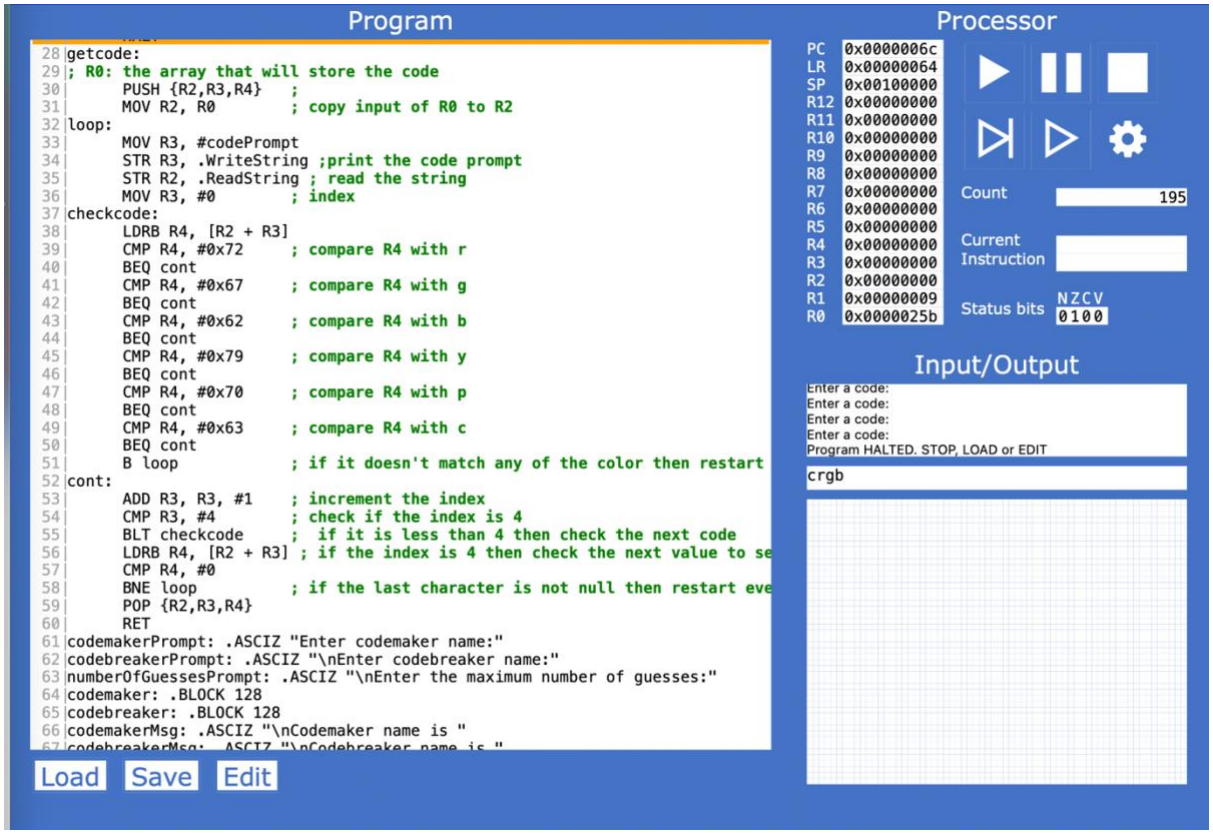
In the first stage, I created multiple labels including the prompts that ask for the user inputs as well as 2 labels to store the string of the code maker and code breaker name. To print the prompts to the screen, I moved the address of the message to Register 0 (R0) first, then store R0 in the pre-defined label called ".WriteString". To get the user input, I use ".ReadString" for strings and ".InputNum" for a number. The code breaker and code maker's names are stored in the labels "codebreaker" and "codemaker" respectively, while the maximum number of guesses is stored in R1. After receiving the user inputs, the program also prints messages to the screen to let the user know what they have entered.



Stage 2:

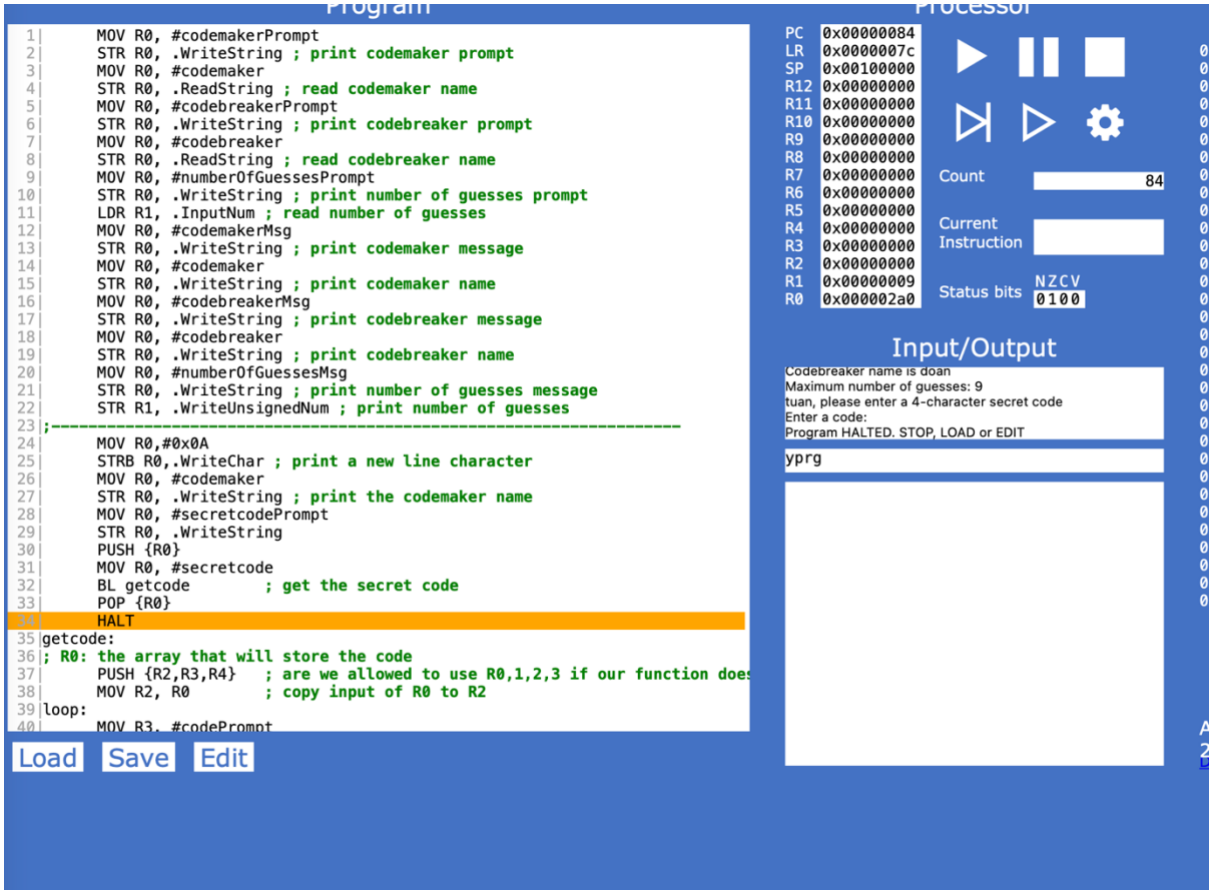
In this stage, I defined a function called "getcode" that reads in 4-character colour code from the user and store it in an array. The array to store the string in is passed in as a parameter to the function. Because this function follows the ABI convention, the input will be stored in R0. Moreover, because this function uses R2, R3, and R4, I pushed those Registers on the stack before using them and pop them back when the function ends. Firstly, the function prints a prompt to ask the user to enter their code and then stores the input code in R2. The next step to do is to check if the input satisfied the requirement that each character is only one of 'r', 'g', 'b', 'y', 'p' or 'c', and the total number of characters is exactly 4. To do this, I used Indexed Memory Addressing technique to iterate through the array and check if each character is one of the allowed characters by using ASCII codes. If the character is allowed, I incremented the index and move on to the next character. However, if there is at least one character that is not 'r', 'g', 'b', 'y', 'p' or 'c', this whole process will restart and the program will ask the user for a new code. Moreover, the function also has to check the

number of characters in the input string. To do this, I compare the index with 4 every time the loop iterates. If the index is less than 4, then the loop iterates again; if the index is 4, then I check if the current character is null. If the character with the index 4 is null, it means that the code has 4 characters, otherwise, the function will restart and ask the user for a new code.



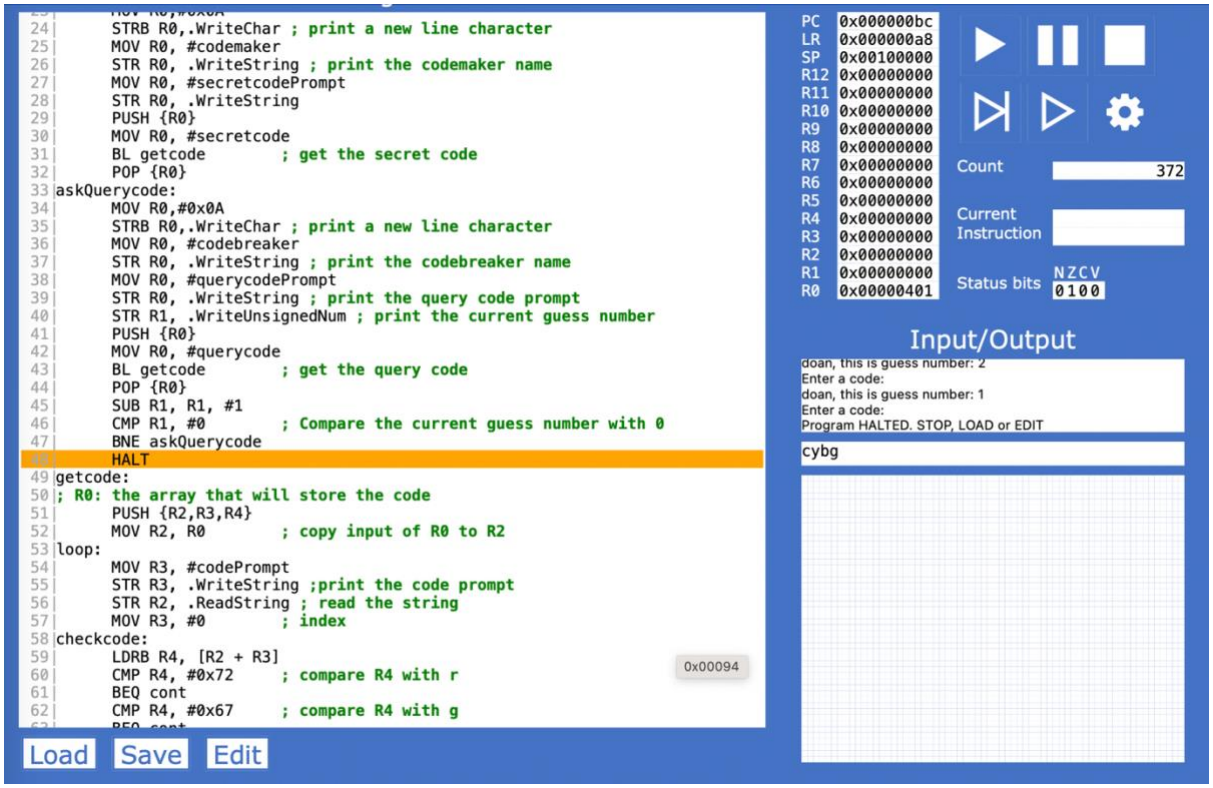
Stage 3:

In this stage, I used the function "getcode" defined in stage 2 to get the secret code from the code maker. Firstly, I defined a label for the secret code and print out a prompt to ask the code maker to enter their secret code. Next, to call the "getcode" function, I pushed the previous value of R0 to the stack, and then move the memory address of "secretcode" to R0, Next, I called the "getcode" function using “BL getcode” command. Finally, I pop the previous value of R0 back.



Stage 4:

In the next stage, the program allows the code breaker to enter their guesses. Firstly, I defined an array "querycode" that will store the code breaker's guesses. Next, the program prints out some messages to ask the user to enter the guess and call the "getcode" function to get the user input in a similar way to stage 3. Notably, the maximum number of guesses is currently stored in R1 back in stage 1. Each time the codebreaker enters a code, the program will decrease R1 by 1 and compare R1 with 0. If R1 is not equal is 0, the program will ask for a new code again. If R1 is 0 then it means that the user has entered the maximum number of guesses and the program will stop.



Stage 5:

Stage 5a:

In the next stage, the program compares the secret code and the query code. The parameters will be stored in R0 and R1 according to the ABI convention. The functions also push the registers that it will use before changing them and pop the values back before the function terminates. There are 3 cases that may occur:

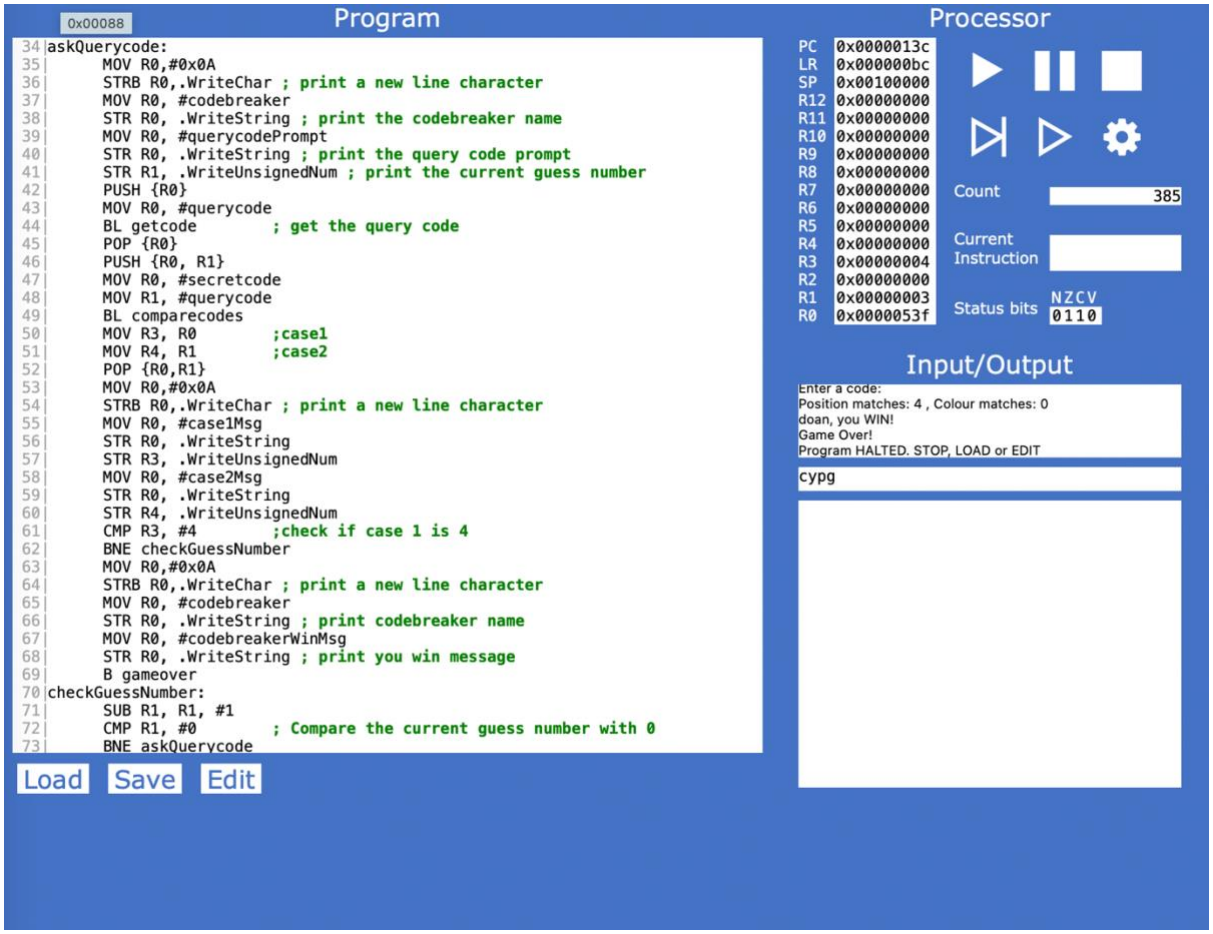
- Case 1: the query peg is a direct match with its corresponding peg in the secret code, OR
- Case 2: the query peg is a match with at least one other peg in the secret code but not in the correct position, OR
- Case 3: the peg has no match in the secret code

This function will return the number of case 1 in R0, and the number of case 2 in R1. To find the number of case 1 and 2, I iterate through the secret code and for each peg I check if the query code has the same colour at the exact position, if that's true then I increment case 1 by 1, otherwise, I iterate through the "querycode" to find if the querycode has that colour but in a different location and increment case 2 by 1 if that's true. If there is no match in the "querycode", I move on to the next character in the secret code. When the index for the secret code reaches 4, the function will terminate.

```
 90 comparecodes:
 91 ;R0 secretcode, R1 querycode
 92       PUSH {R2,R3,R4,R5,R6,R7}
 93       MOV R2, R0         ; copy secretcode to R2
 94  0x0015c R3, R1         ; copy querycode to R3
 95       MOV R0, #0         ;case 1
 96       MOV R1, #0         ; case 2
 97       MOV R4, #0         ; index for secret code
 98 iterateSecretcode:
 99       LDRB R5, [R2 + R4] ; get secretcode peg at index
100       LDRB R6, [R3 + R4] ; get querycode peg at index
101       CMP R5, R6         ; check the pegs
102       BNE else
103       ADD R0, R0, #1     ; increment case 1
104       B incrementSecretcodeIndex
105 else:
106       MOV R7, #0         ; index for query code
107 iterateQuerycode:
108       LDRB R6, [R3 + R7] ; get querycode peg at index
109       CMP R5, R6
110       BNE incrementQuerycodeIndex
111       ADD R1, R1, #1     ; increment case 2
112       B incrementSecretcodeIndex
113 incrementQuerycodeIndex:
114       ADD R7, R7, #1
115       CMP R7, #4
116       BNE iterateQuerycode
117 incrementSecretcodeIndex:
118       ADD R4, R4, #1
119       CMP R4, #4
120       BNE iterateSecretcode ; stop if index reaches 4 (should we check it wi
121       POP {R2,R3,R4,R5,R6,R7}
122       RET
123 codemakerPrompt:  ASCIZ "Enter codemaker name:"
```

Stage 5b

Finally, using the returned value of the "comparecodes" function, I print a message to let the user know how many position matches (case 1) and colour matches (cases 2) there are. Moreover, every time the user enters their guess, I check the position matches to see if it is 4. If that is true, I print a message “you WIN” to the screen and end the program with a “Game Over” message. If the position matches are not 4, I checked the current guess number stored in R1. If the current guess number is 0, then the codebreaker loses the game and a “Game Over” message is also printed out on the screen:

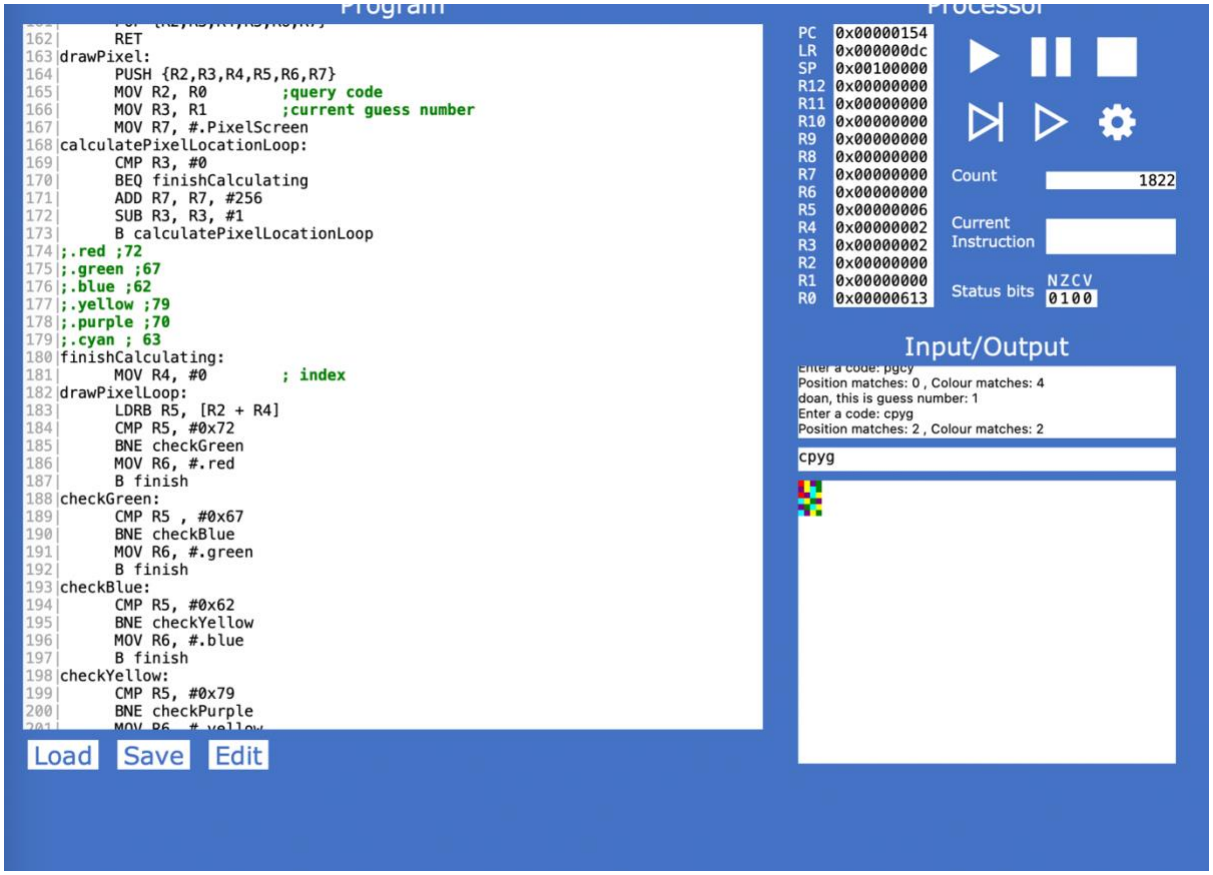


Stage 6:

In this stage, every time the codebreaker enters their guess, a 4-pixel horizontal line will be drawn where each pixel represents the colour in the query code. To do this, I made a new function called drawPixel that accepts the query code and the current guess number as the parameters. Notably, the current guess number is not the number of guesses left. For example, if there are 4 out of 6 guesses left then the current guess number is 2. To calculate the current guess number, I subtract the number of guesses left from the total maximum number of guesses and pass it to the function.

The first thing that the function do is to calculate the location for the start of the line that the 4 pixels will be drawn. This can be calculated by this formula: location = .PixelScreen + 256 * N (where .PixelScreen is the location of the top-left pixel and N is the current number of guesses). I multiply 256 with N because there is 64 pixel in a row and each pixel is 4 bytes. To calculate the location, I use a loop and repeatedly add 256 to ".PixelScreen" and subtract the current number of guesses until it reaches 0.

The next step is to draw the colour. To do this, I iterate through the pegs in the query code and move the corresponding colour to a register. Finally, I store the colour in the location that I previously calculated and increment the location by 4 for the next colour. I also checked the index when it reaches 4 to terminate the function.



Assumptions

The functionalities are implemented based on the requirements and there are no assumptions have been made.

Unresolved Problem

There is no unresolved problem in this program. All functionalities have been tested and the program works according to the requirements.